# OnTerminationPolicy for Tasks and Executors

(Draft not a contribution)

| | |
|---|---|
| Authors | Anand Mazumdar, Megha Sharma, Yan Xu |
| Revision | 0.3 |
| Last Revision | 0.2 |
| Status | Public |

Epics: MESOS-3544, MESOS-3545

Document history

- This document now covers different types of policies on termination, e.g., shutting down the executor, even though restarting tasks on termination remains a main topic of the doc.
- Restartability of a task as proposed in this design is now not a first class concept but rather one type of actions that the framework could choose from to be taken upon the task/executor's termination.

# Motivation

Normally when a task terminates, the framework relies on the status updates sent to the scheduler to act on the terminated task (e.g., relaunch it somewhere else, kill dependent tasks, etc.).

Today when an executor terminates with live tasks (pending launch or launched) on it, all these tasks are killed.

There are scenarios wherein when tasks terminate, requiring a round-trip to the scheduler for such decisions on them are either infeasible or suboptimal.

Such scenarios include:

- Agent gets partitioned from the master so the reaction from the scheduler can be delayed (indefinitely).
- It's expensive to discard container state for the scheduler to relaunch the task somewhere else.
- Frameworks expect sidecar tasks in a task group to be killed when the main tasks die.
- Frameworks expect the executor to be shut down when the certain tasks on it die.

In these scenarios it would be beneficial to handle task/executor termination **locally**, without any involvement from the master and the scheduler.

OnTerminationPolicy is introduced for this purpose: locally handle task/executor termination (not including when the task is explicitly killed)

# Framework APIs

## OnTerminationPolicy

```
```

```
// Modeled after
https://github.com/google/google-http-java-client/blob/master/google-http-client/src/main/java
/com/google/api/client/util/ExponentialBackOff.java
message ExponentialBackoff {
  optional double initial_interval_seconds = 1 [default = 1.0];

  optional double max_interval_seconds = 2 [default = 300.0];

  // Potentially for generic use cases (outside the scope of this work).
  // optional double randomization_factor = 3 [default = 300.0];
  // optional double max_elapsed_time = 4;
  // optional double multiplier = 5;
}

// Describes how an entity (currently task or executor) should be restarted
// by the entity that launches it.
message RestartPolicy {
  // The first restart will be immediate and subsequent restarts are backed off
  // according to the backoff policy (default values already specified)
  optional ExponentialBackoff restart_backoff = 1;

  // Reset the restart backoff after the launched entity has been running
  // for some amount of time. Default to 10 minutes.
  optional double reset_backoff_seconds = 4 [default = 600.0];
}


message TaskInfo {
  // ...

  message OnTerminationPolicy {
    // PROCEED: Proceed with task termination.
    // SHUTDOWN_EXECUTOR: Shut down the executor and kill all live tasks.
    // RESTART: See 'Restart' below.
    enum Type {
      UNKNOWN = 0;
      PROCEED = 1;
      SHUTDOWN_EXECUTOR = 2;
      RESTART = 3;
```

```
      // TODO:
      // KILL_TASK_GROUP = 4;
      // RESTART_TASK_GROUP = 5;
    }

    // Action to restart the task. Implemented for executor-less command-based
    // tasks. For tasks that specify an executor, it is the executor's
    // responsibility to implement the action.
    message Restart {
      enum Type {
        UNKNOWN = 0;
        ALWAYS = 1;  // Always restart the task.
        ON_FAILURE = 2;  // Restart the task when it has failed.
      }

      optional Type type = 1; // No need for [default = ALWAYS]?

      optional RestartPolicy restart_policy = 2;
    }

    // Action to shutdown the executor. Implemented for the built-in default executor. For
   // tasks that specify an executor, it is the executor's responsibility to implement the
   // action.
    message ShutdownExecutor {
      enum Type {
        UNKNOWN = 0;
        ALWAYS = 1;  // Always shutdown the executor.
        ON_FAILURE = 2;  // Shutdown the executor when the task has failed.
      }

      optional Type type = 1;
    }

    optional Type type = 1;

    // Needs to be set if `type` is `RESTART`.
    optional Restart restart = 2;

    // Needs to be set if `type` is `SHUTDOWN_EXECUTOR`.
    optional ShutdownExecutor shutdown_executor = 3;
  }

  optional OnTerminationPolicy on_termination_policy = 14;
}


message ExecutorInfo {
  //...
```

```
    message TerminationPolicy {
      // RESTART: Restart the executor if there are any live restartable
      //          tasks; otherwise proceed with termination.
      enum Type {
        UNKNOWN = 0;
        RESTART = 1;
      }

      message Restart {
        // Policy for restarting the executor and relaunch the tasks.
        // If not set, a restart policy with default values are used.
        optional RestartPolicy restart_policy = 1;
      }

      optional Type type = 1;

      optional Restart restart;
    }
  }

  optional OnTerminationPolicy on_termination_policy = 15;
}
```

## Notes

- A “restartable task” is not a first-class concept but rather is loosely defined as the task with `task.on_termination_policy.type == RESTART` (or `RESTART_TASK_GROUP`).
- A “restartable task group” is a task group wherein all tasks are “restartable” (as defined above).
- Command tasks don’t have ExecutorInfo and Mesos will to set the command executor’s on termination policy type to be `` `RESTART` `` if the task’s on termination policy type is `` `RESTART` ``.
- If a scheduler kills a task explicitly via `killTask()`, any set `OnTerminationPolicy` is ignored.
- There are no framework capabilities to set, as usage of this feature is initiated by the scheduler, at the task and executor level.
- OnTerminationPolicy is defined on both the task and the executor, which governs the task’s and the executor’s behavior on termination, respectively.
- The container’s lifecycle is bound to the executor so when tasks terminate they are restarted (if a restart policy is used) in the same container whereas when the executor is restarted, tasks are relaunched into the container owned by the new executor instance.
- Note that `` `ExecutorInfo::OnTerminationPolicy::RESTART` `` means the executor is restarted “when necessary”: there are alive tasks that are restartable.

## OnTerminationPolicy Usage Examples

Explained below are the various task/taskgroup/executor shutdown/restart scenarios that can be achieved with the OnTerminationPolicy.

### To restart the task and its executor

The scheduler wishes to have the task restarted by its executor when it terminates and have the executor restarted when it (the executor) terminates.

Consistent with other Mesos APIs that expose a union type, it can set:

```
  task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
  task.mutable_on_termination_policy()->mutable_restart();
  task.mutable_executor()->mutable_on_termination_policy()->set_type(
      ExecutorInfo::OnTerminationPolicy::RESTART);
  task.mutable_executor()->mutable_on_termination_policy()->mutable_restart();
```

Note non-restartable tasks are not relaunched upon executor restart.

#### Discussion

If TaskInfo::OnTerminationPolicy::Type doesn't have a default, then additionally the framework needs to specify:

```
  task.mutable_on_termination_policy()->mutable_restart()->set_type();
      TaskInfo::OnTerminationPolicy::Restart::ALWAYS);
```

### Maintaining old behavior: proceed with task/executor termination.

Executors are expected to maintain the old behavior when OnTerminationPolicy is not specified. This is not prescribed by the API but for most executors the following should be equivalent to not setting OnTerminationPolicy.

```
  task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::PROCEED);
  task.mutable_executor()->mutable_on_termination_policy()->set_type(
      ExecutorInfo::OnTerminationPolicy::PROCEED);
```

However note that for the built-in default executor though, not having an `OnTerminationPolicy` for a task  is equivalent to (its current behavior):

```
task.mutable_on_termination_policy()->set_type(
    TaskInfo::OnTerminationPolicy::SHUTDOWN_EXECUTOR);

task.mutable_executor()->mutable_on_termination_policy()->mutable_shutdown()->set_type(
    TaskInfo::OnTerminationPolicy::ShutdownExecutor::ON_FAILURE);
```

### Restart the executor (which relaunch live tasks) but don’t restart tasks by the executor.

It’s possible that 1) the task cannot tolerate a dirty container (e.g., sandbox) or 2) the custom executor doesn't implement support for restarting tasks.

```
  task.mutable_executor()->mutable_on_termination_policy()->set_type(
      ExecutorInfo::OnTerminationPolicy::RESTART);
  task.mutable_executor()->mutable_on_termination_policy()->mutable_restart();
```

Then the agent will restart the executor.

### Restart the task only by the executor

It’s possible that it only makes sense to restart the task by the running executor (e.g., it’s a task within a task group or it relies on the sandbox state).

```
task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
task.mutable_on_termination_policy()->mutable_restart();
```

Then the task is restarted by the running executor but when the executor terminates, the agent doesn’t restart the executor (and the tasks are killed).

### Restart a task in a task group by the running executor

In a task group it’s desirable to keep the sidecars up as long as the main tasks are up.

For the sidecar tasks:

```
task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
```

```
task.mutable_on_termination_policy()->mutable_restart();
```

No changes needed for the main tasks but when the main tasks die, **it's upon the executor to decide what happens when tasks with on termination policy specified dies**.

### Restart the executor when it terminates and relaunch task groups on it.

```
launchGroup.mutable_executor()->mutable_on_termination_policy()->set_type(
    ExecutorInfo::OnTerminationPolicy::RESTART);
launchGroup.mutable_executor()->mutable_on_termination_policy()->mutable_restart();

// And, for all tasks in the group, use RESTART or RESTART_TASK_GROUP.
task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
task.mutable_on_termination_policy()->mutable_restart();
```

### (Invalid) Restart the executor when it terminates and relaunch some live tasks in a task group on it.

If **not all** tasks in a task group have `TaskInfo.on_termination_policy.type == RESTART` but `ExecutorInfo.on_termination_policy.type == RESTART`, it's an invalid config because the executor upon restart cannot just relaunch some of the tasks in the group - it violates the "all or nothing" principle.

### (TODO) Restart the executor with task groups on it and restart the task group when one task in it terminates

```
  launchGroup.mutable_executor()->mutable_on_termination_policy()->set_type(
      ExecutorInfo::OnTerminationPolicy::RESTART);
  launchGroup.mutable_executor()->mutable_on_termination_policy()->mutable_restart();

  // Tasks for which you want to restart the entire task group when it terminates.
  task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART_TASK_GROUP);

  // Tasks for which you just want to restart itself.
  task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
```

Once we implement `TaskInfo.on_termination_policy.RESTART_TASK_GROUP` we can require all tasks in a group to either have the type `RESTART_TASK_GROUP` or `RESTART` for the restarted executor to relaunch the group and thus still maintain the “all or nothing” semantics.

### Kill the executor when the main tasks in a task group terminate

In these cases we want to shutdown the executor when main tasks terminate.

```
// For main tasks:
task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::SHUTDOWN_EXECUTOR);

// For sidecar tasks, either don’t set OnTerminationPolicy or to type to RESTART.
task.mutable_on_termination_policy()->set_type(
      TaskInfo::OnTerminationPolicy::RESTART);
task.mutable_on_termination_policy()->mutable_restart();
```

### (TODO) Restart a task group when the main task terminates

In certain cases we don’t want to restart the sidecar tasks but we also don’t want them to bring down the executor/task group. We only want “main” tasks to do that.

```
// For main tasks:
task.mutable_on_termination_policy()->set_type(
    TaskInfo::OnTerminationPolicy::RESTART_TASK_GROUP);

// For sidecar tasks, either don’t set OnTerminationPolicy or to type to RESTART.
task.mutable_on_termination_policy()->set_type(
    TaskInfo::OnTerminationPolicy::RESTART);
task.mutable_on_termination_policy()->mutable_restart();
```

Behavior: The terminated main tasks kill the executor and all other tasks (including other main tasks) while the sidecars don’t.

## TASK_RESTARTING

New state introduced for `TaskInfo::OnTerminationPolicy::Restart`:

- To inform the schedulers about the task restart.
- Sent by the agent for relaunched tasks after the executor restart.
    - It may be followed by a TASK_STARTING sent by the executor.

    - No executor involvement in this case.
- The executor should send a TASK_STARTING for the task it restarted.
- The scheduler is supposed handle TASK_STARTING if it sends tasks with `TaskInfo::OnTerminationPolicy::Restart`.
- It is a regular non-terminal state.

# Upgrades

- Old Frameworks don't see this behavior because you have to set the OnTerminationPolicy explicitly.
- Old master is expected to handle the new state gracefully.
- Upgrading the agent or the master first should both be OK.
- Rolling back the agent should be straightforward.
    - Old agent just ignores the checkpointed TaskInfo and the OnTerminationPolicy in the tasks launched by a new master.

# Implementation details for restartable tasks

## Support executor restart

- **Current behavior**:
    a. **Running Agents:** If the executor terminates while the agent is running, the agent destroys the container and transitions all tasks to a terminate state.
    b. **Recovering Agents:** During recovery, the agent sends a reconnect request to all the executor pids it knows about from the checkpoint and if they have already terminated then the respective executors don't re-register with the agent in the agent's re-register timeout and therefore these executors are deemed terminated.

- **Proposed change**:
    a. **Running Agents**: For agents which are up, upon receiving the executor lost/termination notification they will examine the `executor.on_termination_policy.restart` on the terminated executor to determine if it needs to restart the executor.

    b. **Recovering Agents:** For an agent which is in recovery, the best time to restart the restartable executor is at the end of agent's recovery. During recovery the agent sends reconnect request to all the executor pids it knows from the checkpoint. For the executor pids which have already terminated the respective executors don't re-register with the agent in the agent's re-register timeout. The restart policy for these terminated executors is then examined by the agent to determine if they are a candidate for restart.

c. The executor with no non-terminal tasks will not be considered for restart.

d. The individual tasks in the TaskGroup will be evaluated to determine the restart candidacy of the TaskGroup. The executor will be restarted with the entire TaskGroup if it's eligible.

## Allow agent to recover after host reboot

In order to restart executors terminated due to host reboot, the agent will be allowed to recover its state which includes its SlaveID, the tasks/executors it was running prior to the reboot and re-register with the master. This way a recovering agent can determine which executors need to be restarted and restart them at the end of its recovery.

https://issues.apache.org/jira/browse/MESOS-6223

1. **Current behavior**: When the agent host has not rebooted and the agent process restarts which is the first case, it recovers its own state which includes its `SlaveID` and the state of all the frameworks/executors/tasks etc running on it so it can reconnect with the old executors which might still be running and re-register with master using the old `SlaveID.` But when the agent is recovering post a host reboot then it short circuits the recovery and doesn't attempt to recover any of the aforementioned state and as a result attempts to register with the master to get a new `SlaveID.` During recovery, the agent compares the checkpointed `boot_id` with the system's current `boot_id` to determine if there's been a reboot.

2. **Proposed change**: In order to support restart of tasks during agent recovery, the agent needs to be able to recover from checkpoint and keep its old `SlaveID` and re-register with the master. So, we will force the agent to go through the recovery post a reboot as well.

### Checkpointing TaksInfo/TaskGroupInfo for launching tasks

1. **Current behavior**: The agent only checkpoints Task proto derived from TaskInfo which doesn't have enough information to launch a task. The agent also doesn't checkpoint the TaskGroupInfo which is needed to launch the TaskGroup.

2. **Proposed change**: To support task/task group restart post agent restart TaskInfo and TaskGroupInfo are essential to be checkpointed. For protobuf backwards compatibility, in the new version we can checkpoint both `Task` and `TaskInfo` side by side since they are serving different purposes. Later, we could consider stopping to checkpoint Task. During recovery we can first try to read from the TaskInfo checkpoint, if not available, we read from the Task checkpoint and in the next version we stop checkpointing Task.

### Launching the Restartable executors

1. **Proposed change:**

a. A Running agent will call the runTask(..)/runTaskgroup(..) methods to restart an eligible executor upon receiving the executor termination/lost notification.

b. A Recovering agent will look at the onTerminationPolicy.restartPolicy on executors that didn't re-register within the re-register timeout to find the eligible candidates for restart.

c. The agent is in the state `RECOVERING` during its recovery. The agent doesn't attain the state `RUNNING` until it has registered with the master and any `runTask()/runTaskgroup()` calls to the agent will be rejected if agent's state is not running. The agent also verifies the `PID` from which it is receiving the run task calls and doesn't act if its coming from any entity other than the master. We will change the method to allow task launch when the agent is diconnected or recovering. We will change the function to allow the agent to run tasks while it is `RECOVERING or DISCONNECTED.` The agent will also be made to accept `runTask()` calls from itself and the master.

d. The relaunched executor will have all its Tasks/TaskGroups restarted in new containers (if the tasks in the TaskGroup have been using containers to begin with) so, the old sandboxes for the tasks will not be reused and they will need to refetch the URIs/images etc if they are using one.

e. The executor will be restarted with restartable Tasks/TaskGroups.

f. The agent will send a TASK_RESTARTING status update instead of TASK_FAILED/TASK_LOST to let the frameworks know that the agent has restarted the executor and the frameworks can look at the source to know which entity restarted the task and take appropriate action.

g. The tasks which are not to be restarted will have their respective sandboxes scheduled for garbage collection.

## Support Task/TaskGroup restart by executor

The executor, if alive, is responsible for performing the task/taskgroup restart.

### Candidate for restart

1. **Proposed Solution:** The executor upon observing a terminated task will determine the candidacy of terminated tasks based on their RestartPolicy and only the tasks marked restartable (by the executor) will be restarted.

### Launching the Tasks

1. **Proposed Solution:** The executor will launch only the eligible tasks (even for TaskGroups) and send a TASK_RESTARTING status update for the restarted task, followed by the usual status updates. The tasks restarted by the executor reuse the old container and hence the sandbox.

### Reconciliation with Master

1. **Current behavior:** No changes are required to be made on the master's end to support restartable tasks. Master allows the rebooted agent to re-register and reconcile the state of tasks/executors running on it including the ones which have been restarted by the agent (Restartable tasks). See Annexure for details.

### Garbage Collection

Currently, when the agent reboots then the entire old agent's root directory is scheduled for garbage collection (since it registers with the master as a new agent). For all tasks we will let the agent schedule the framework, executor and run paths for garbage collection and when the `runTask()` is called on the agent then it will automatically unschedule the corresponding framework and executor directories for the revivable task.

# Interaction With Other Features

## Restartable Tasks and Partition Aware Frameworks

- If the master has deemed the agent as unreachable while it was gone, then upon coming back all the tasks on the agent which have been launched by partition un-aware frameworks will be killed by the master.
    - In order to prevent the master from killing the tasks which have been restarted by the agent, the frameworks need to opt in to partition awareness capability.
- TASK_RESTARTING is just a regular non-terminal state and its transitions from and to task states introduced by partition awareness obey other rules followed by states like TASK_RESTARTING or TASK_STARTING.

## Framework Failover

Framework failover case is no different than what happens now. Restartable tasks will be killed if the framework doesn't register within the failover timeout.